Having cake and eating it too: painless and efficient cluster utilization for data scientists

Artem Trofimov

High Load++ Armenia

Co-organizer Yandex

1. DataScientists & Hardwate
2. Serverless Jupyter
3. Cluster Injections
4. Optimizations & performance

Typical DS pipeline

1

Data exploration

2

Feature engineering

3

ML model building

4

Deployment

Ways to organize work

Cloud VM & SSH

JupyterHub

Kubeflow

KF Pipelines/Metaflow/etc

…

Managed ML platforms

Resources granularity

Cloud VM & SSH	→	VM
JupyterHub	→	Container
Kubeflow	→	Container/API calls
KF Pipelines/Metaflow/etc	→	Pipeline
…		…
Managed ML platforms	→	VM/Container/Pipeline

VM/Container properties

Manual lifecycle

Complex migration

Inefficient utilization

Utilization in numbers

~12% VMs

~35% Containers

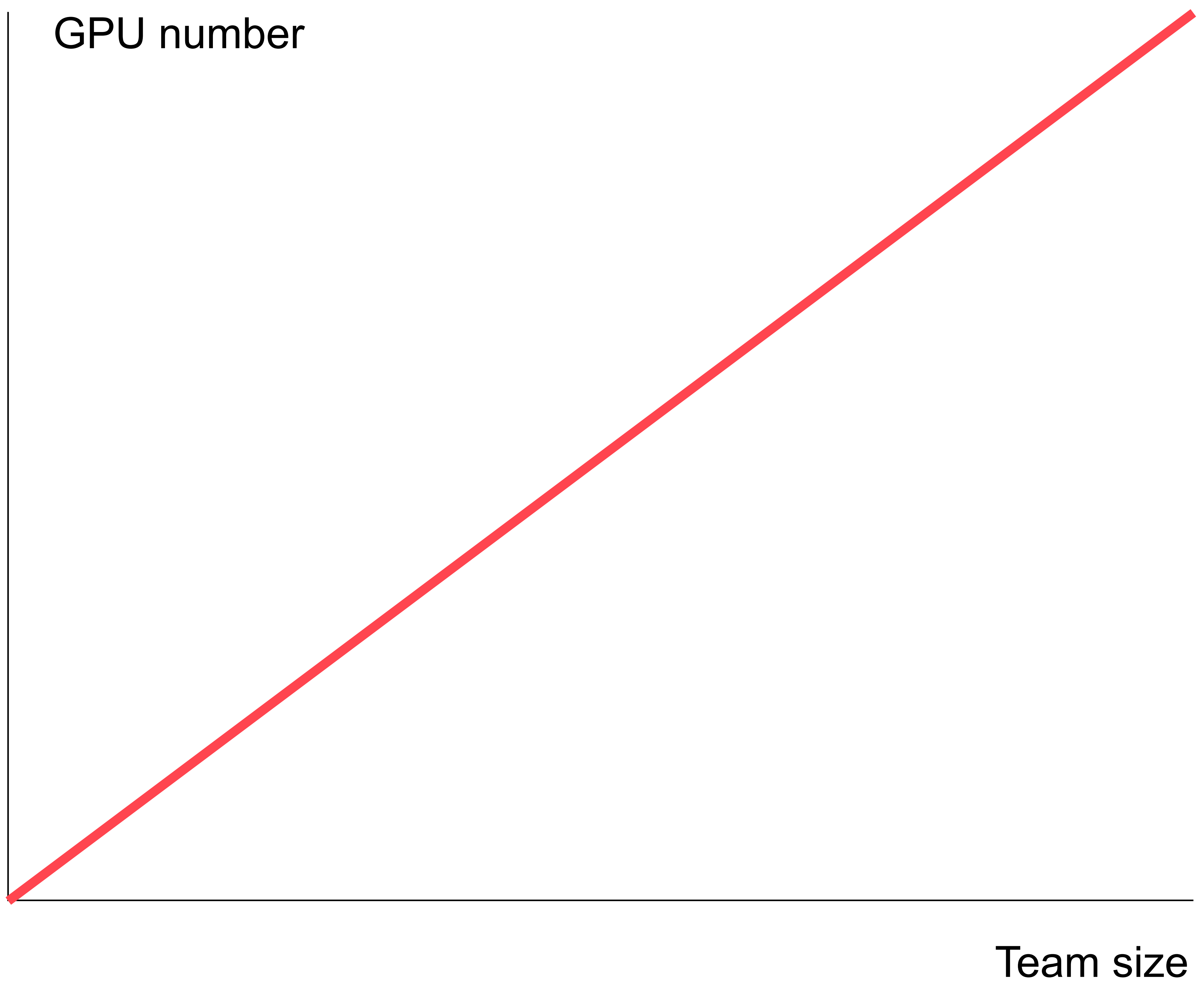
*allocated time/code running time ratio
**data provided by our customers

Inefficient = expensive

4 CPU 32GB RAM	8 CPU 64GB RAM	VERY EXPENSIVE GPU
Data exploration	Feature engineering	Model building

VERY EXPENSIVE GPU per hour × work time (8h)

Scalability issues



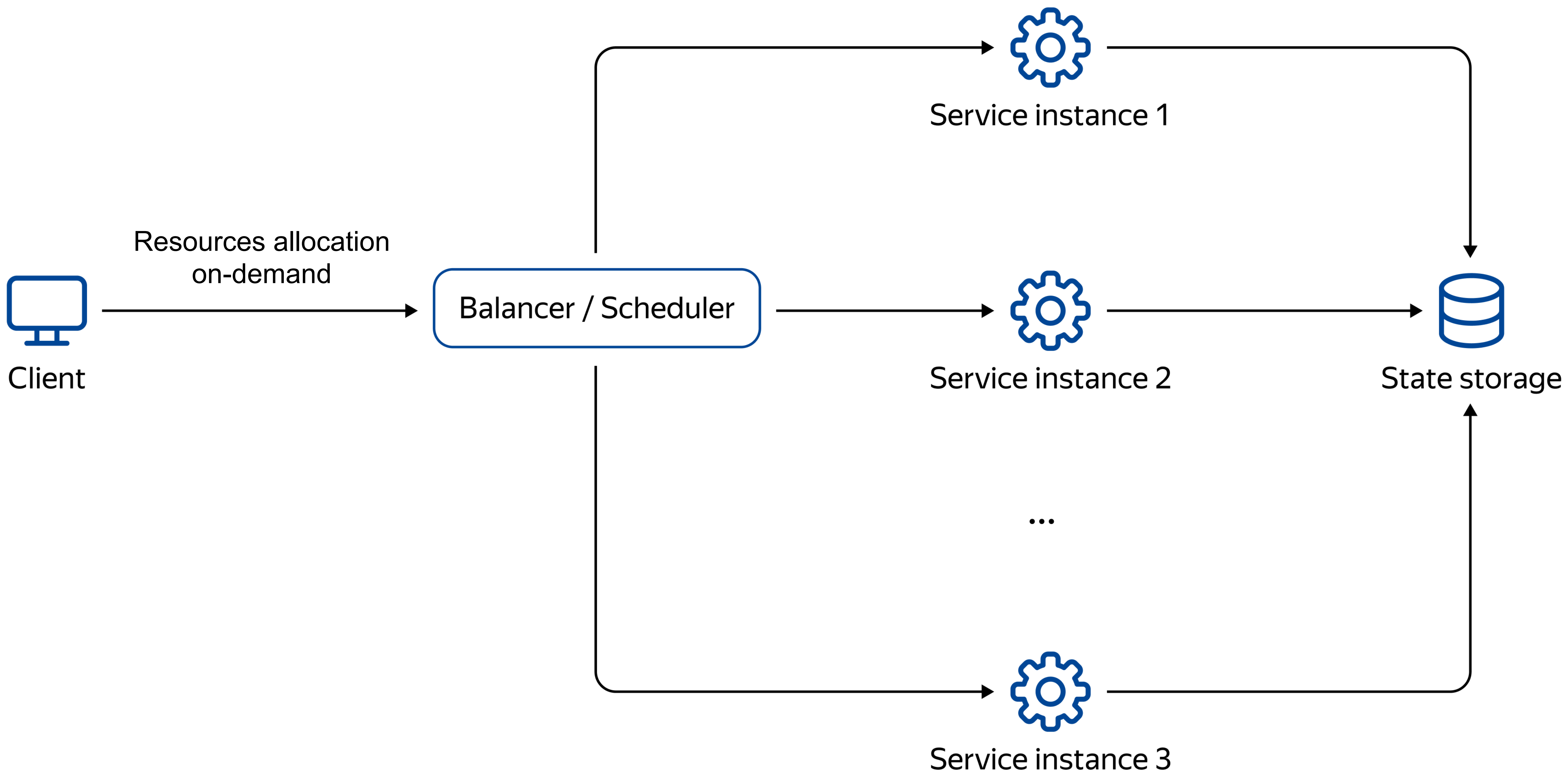
Restriction policies are painful

Work good for small teams

50 Data Scientists hardly can peacefully negotiate

meme-arsenal.com/en/create/template/2831793

Oh, wait. Serverless?



Production service vs data science workflow

Fixed env		Dynamic env (pip instal)
Standard artifacts (docker)		Arbitrary code on a local laptop
State in database	vs	Local state
Should scale out		Should scale up

Serverless for DS is painful

SDK requires heavy code rewriting

Startup time can be slow due to complex env and state

Resources are usually allocated per pipeline

Example

Define a standalone Python function.
This function must meet the following requirements

- It should not use any code declared outside of the function definition
- Import statements must be added inside the function
- Helper functions must be defined inside this function

```
def my_divmod(dividend: float, divisor: float) -> NamedTuple
    # Import the numpy package inside the component function
    import numpy as np

    # Define a helper function
    def divmod_helper(dividend, divisor):
        return np.divmod(dividend, divisor)
```

kubeflow.org/docs/components/pipelines/v1/sdk/python-function-components

Can serverless be painless?

1

Existing code reuse

2

UX comparable with conservative tools

3

Fine-grained resources allocation

1. DataScientists & Hardwate
2. Serverless Jupyter
3. Cluster Injections
4. Optimizations & performance

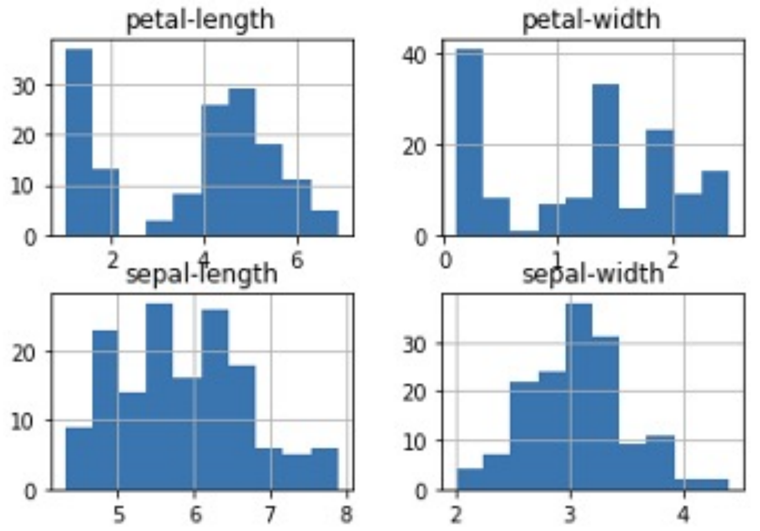
Way #1. Serverless jupyter

Load data

```
[2]: url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
dataset = pd.read_csv(url, names=names)
```

Visualize data

```
[4]: dataset.hist()
plt.show()
```

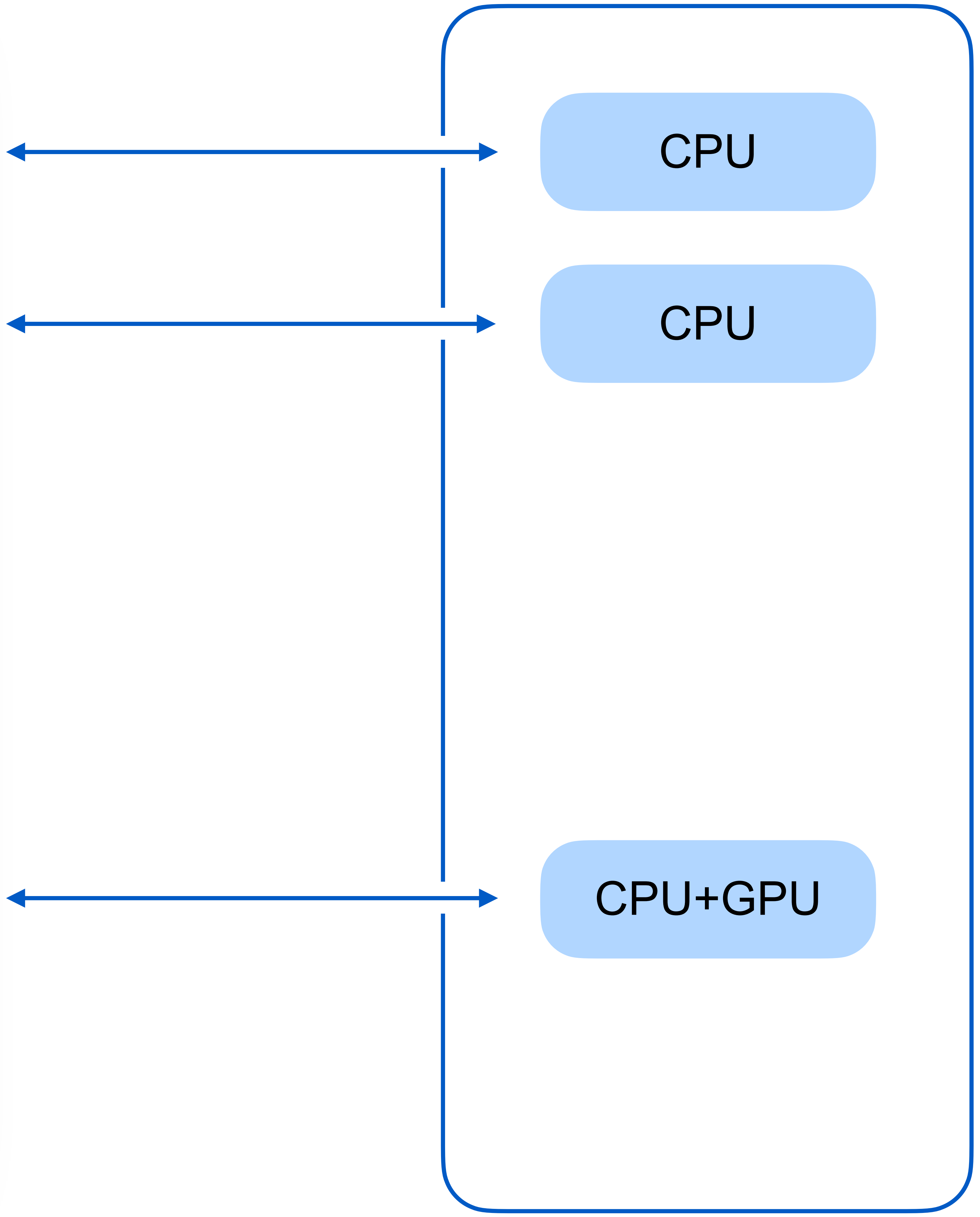


Build model

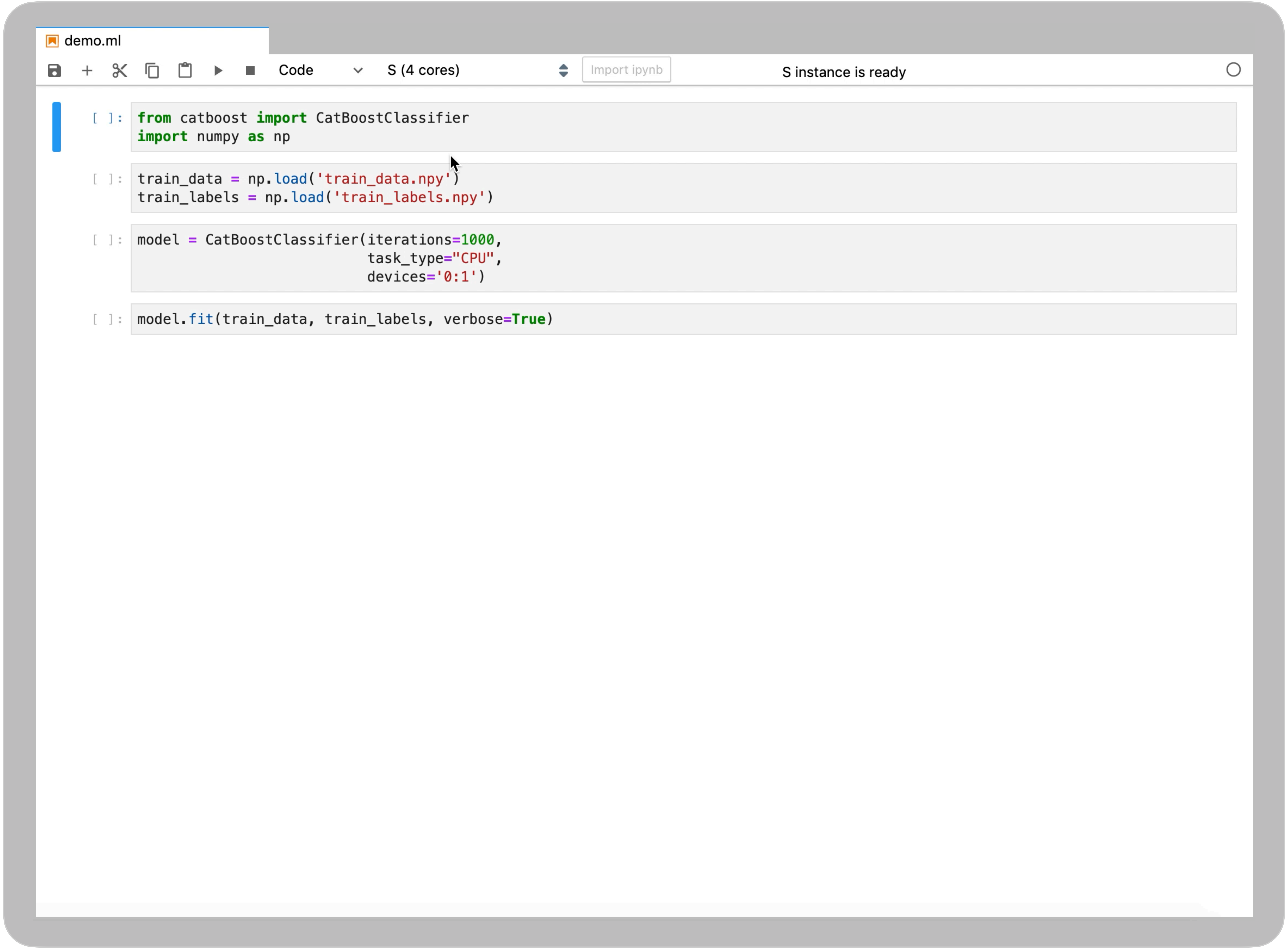
```
[16]: array = dataset.values
X = array[:, 0:4]
Y = array[:, 4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = \
model_selection.train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

```
[17]: kfold = model_selection.KFold(n_splits=10, random_state=seed)
cv_results = model_selection.cross_val_score \
(SVC(gamma='auto'), X_train, Y_train, cv=kfold, scoring='accuracy')
msg = "%s: %f (%f)" % ('Accuracy', cv_results.mean(), cv_results.std())
print(msg)
```

```
Accuracy: 0.991667 (0.025000)
```



Demo

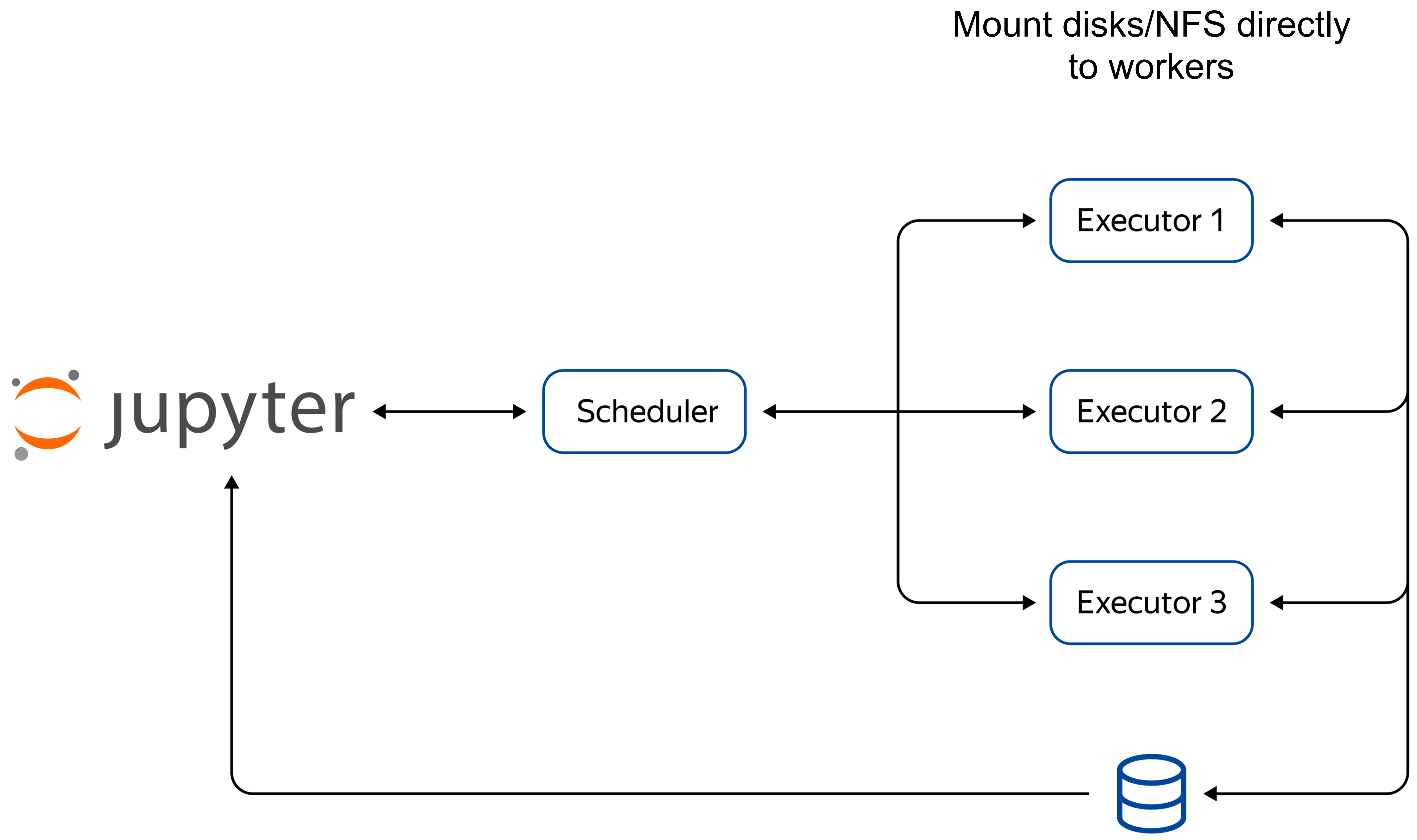

demo.ml
Code
S (4 cores)
Import ipynb
S instance is ready
[ ]: from catboost import CatBoostClassifier
import numpy as np
[ ]: train_data = np.load('train_data.npy')
train_labels = np.load('train_labels.npy')
[ ]: model = CatBoostClassifier(iterations=1000,
task_type="CPU",
devices='0:1')
[ ]: model.fit(train_data, train_labels, verbose=True)

What is the state?

Production service $\longrightarrow$	Structured data in database
Jupyter $\longrightarrow$	FS + python interpreter

State: file system

- Disks/NFS mounted to executors
- No concurrent access



State: python variables

1

Complex structure

2

Can be BIG

Limited by RAM

3

No concurrent access

How to save interpreter state

Serialization (pickle?)

CRIU snapshots

???

Blob storage/Disk

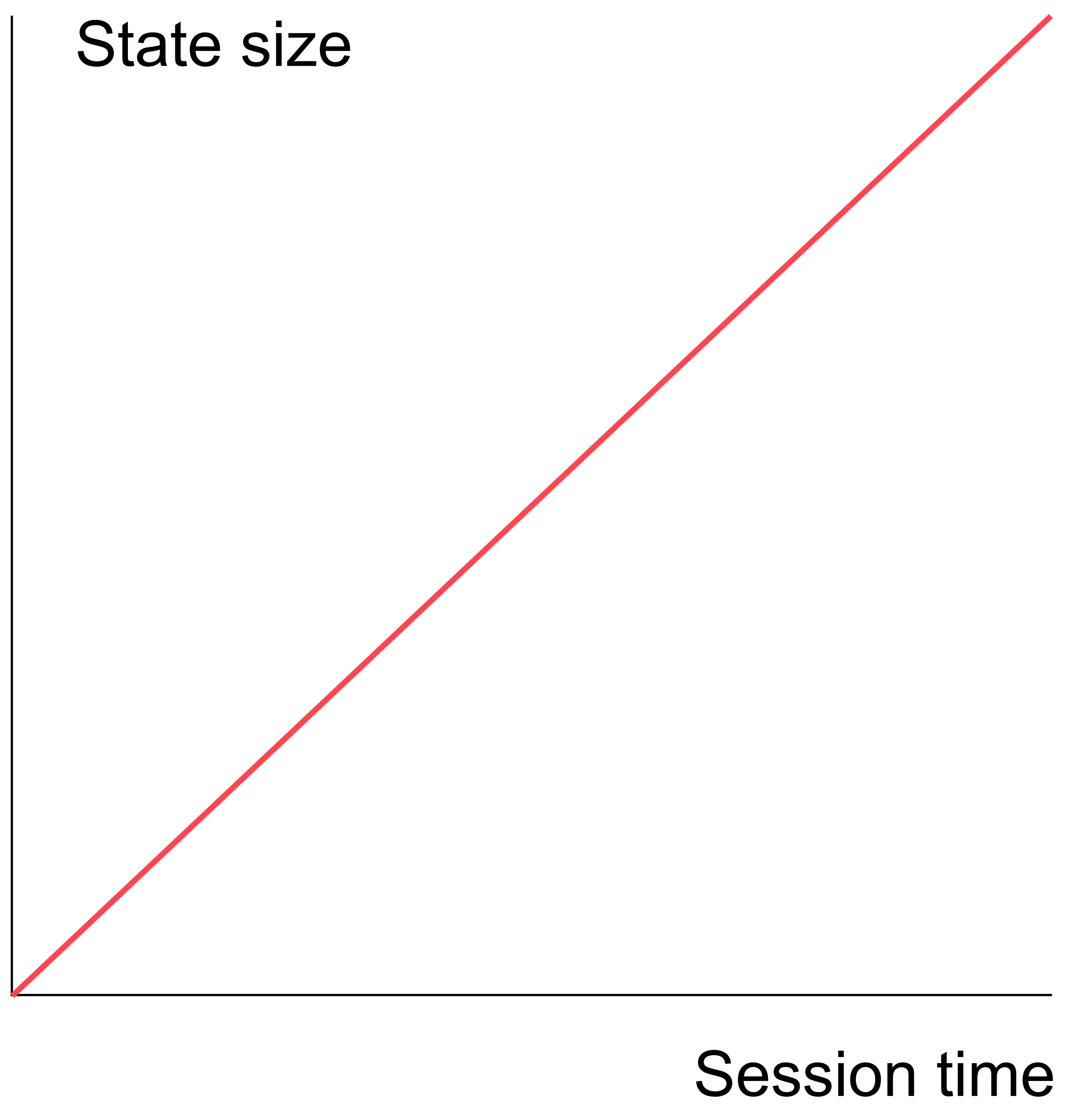
Pitfalls

!

For correct work we need to serialize ALL variables, but most of them are temporal

CRIU snapshots processes and does not support lazy loading

State growth problem



All these variables will be in the state!

```
    return full_val_loss / (overall_sequence_length * 88)

In [8]: clip = 1.0
        epochs_number = 12000000
        sample_history = []
        best_val_loss = float("inf")

        for epoch_number in xrange(epochs_number):

            for batch in trainset_loader:

                post_processed_batch_tuple = post_process_sequence_batch(batch)

                input_sequences_batch, output_sequences_batch, sequences_lengths = post_processed_batch_tu
        ple

                output_sequences_batch_var =  Variable( output_sequences_batch.contiguous().view(-1).cuda
```

conferences.oreilly.com/jupyter/jup-ny/public/schedule/detail/68282

Serverless Jupyter: overview

- Compatible with vanilla Jupyter
- Works pretty good for simple state (popular libs)
- Can be painful for users with complex env

DataSphere — our serverless Jupyter implementation

https://cloudil.co.il

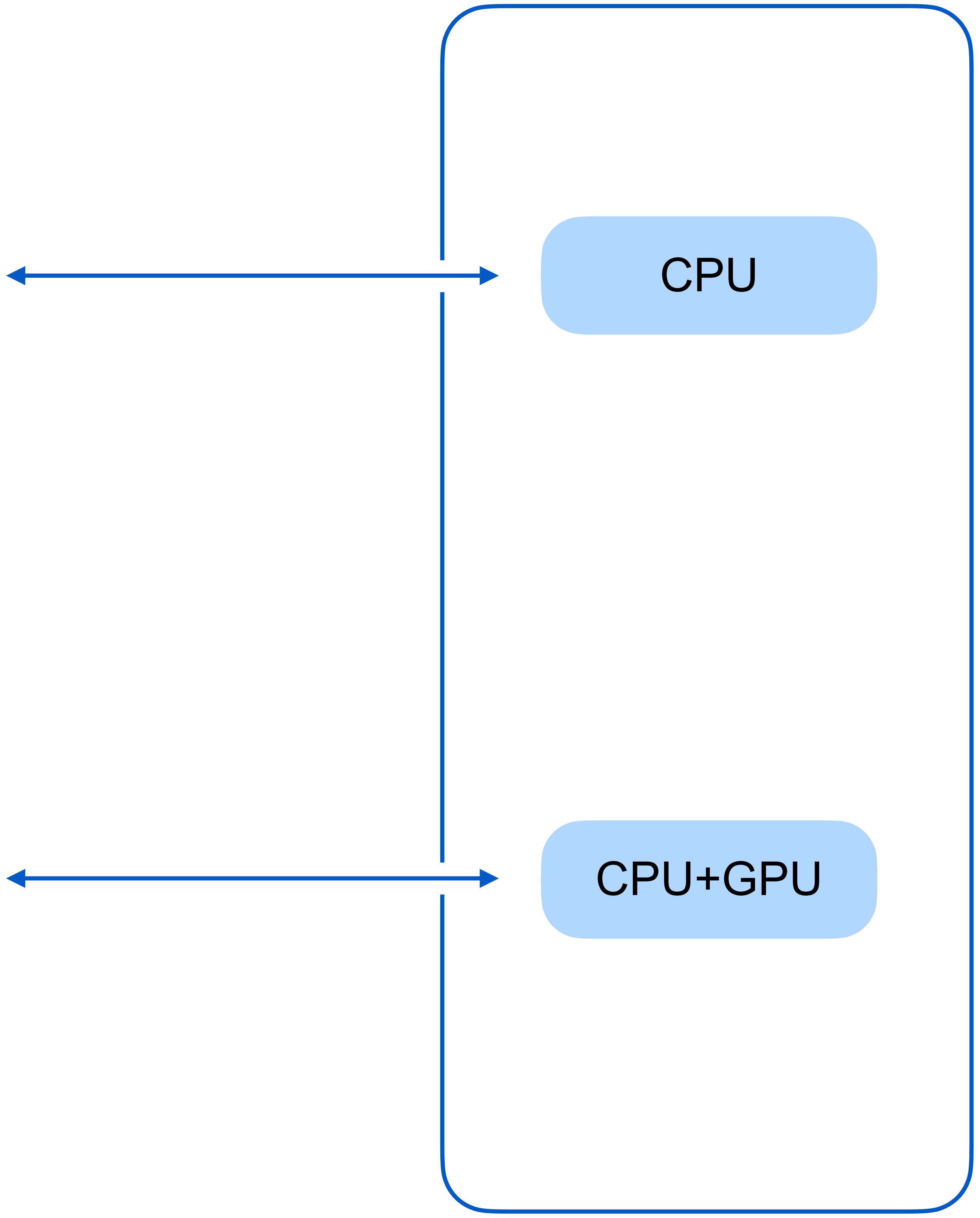
1. DataScientists & Hardwate
2. Serverless Jupyter
3. Cluster Injections
4. Optimizations & performance

Way #2. Cluster injections

```python
@op
def dataset() -> Bunch:
    raw_data = load_breast_cancer()
    data = clean_data(raw_data_set)
    return data_set


@op(gpu=Gpu.any())
def train(data_set: Bunch) -> Classifier:
    model = CatBoostClassifier(
        iterations=1000, task_type="GPU"
    )
    model.fit(data_set.data, data.target)
    return cb_model


data = dataset()
model = train(data)
```



Demo

catboost_whiteboard.py

```python
from catboost import CatBoostClassifier
from lzy.api.v1 import Gpu, LzyRemoteEnv, op
from sklearn import datasets
from sklearn.model_selection import GridSearchCV
from sklearn.utils import Bunch


def dataset() -> Bunch:
    data_set = datasets.load_breast_cancer()
    return data_set


def search_best_model(data_set: Bunch) -> GridSearchCV:
    grid = {"max_depth": [3, 4], "n_estimators": [100, 200]}
    cb_model = CatBoostClassifier(train_dir="/tmp/catboost")
    search = GridSearchCV(estimator=cb_model, param_grid=grid, scoring="accuracy", cv=3)
    search.fit(data_set.data, data_set.target)
    return search


data = dataset()
model = search_best_model(data)
```

dataset()

Cluster injection principles

- No IDE binding
- Minimal changes in existing code

  Not fair for Notebooks
- Automatic environment migration
- Hybrid execution

Function as a computational unit

Operation is an ordinary python function with type annotations

Decorators (meta-information) are used for hardware requirements

Hardware requirements

Type annotations

```
@op(gpu=Gpu.any())
def train(data_set: Bunch) -> CatBoostClassifier:
    cb_model = CatBoostClassifier(iterations=1000)
    cb_model.fit(data_set.data, data_set.target)
    return cb_model
```

State: func args and return values

- Only arguments and return values must be serializable
- Temporal variables live only during execution

No need to serialize all these variables!

```
def solve_lorenz(sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""

    max_time = 4.0
    N = 30

    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

    # Solve for the trajectories
    t = np.linspace(0, max_time, int(250*max_time))
    x_t = np.asarray([integrate.odeint(lorenz_deriv, x0i, t)
                      for x0i in x0])

    # choose a different color for each trajectory
    colors = plt.cm.viridis(np.linspace(0, 1, N))

    for i in range(N):
        x, y, z = x_t[i,:,:].T
        lines = ax.plot(x, y, z, '-', c=colors[i])
        plt.setp(lines, linewidth=2)
    angle = 104
    ax.view_init(30, angle)
    plt.show()

    return t, x_t
```

Automatic env migration

import mylib

Installed from pypi?

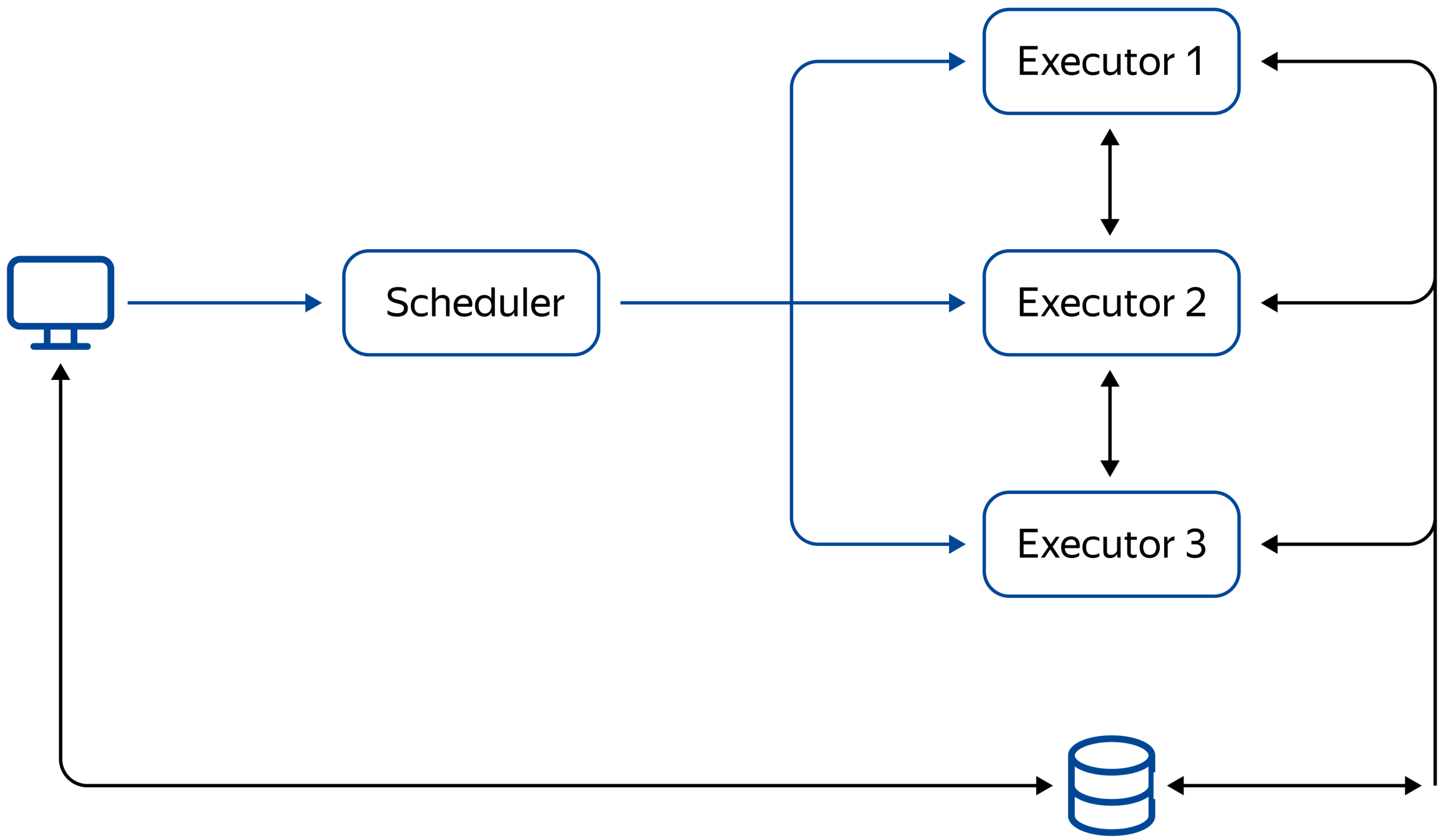
Yes

Add pip install command
to conda.yaml

No

Detect module dirs
and transfer them as archive

Hybrid execution



— Data plane — Control plane

Pitfalls

Complex environments cannot be captured automatically	For complex environments, it is possible to override docker image
Need for code migration from notebooks	Small changes if code is written in procedural style
Local data uploading to storage can be slow	We can cache data during a working session

Cluster injections overview

- No IDE binding
- No environment/OS binding
- Existing code friendly for Jupyter haters
- Can be painful for Jupyter lovers

Λzy — our open-source cloud injections lib over k8s

clck.ru/32sZ8x

1. DataScientists & Hardwate
2. Serverless Jupyter
3. Cluster Injections
4. Optimizations & performance

Optimizations

Huge conda/docker environment	→	Cache on SSD disks (we can run docker with layers on another device)
VMs creation can be slow	→	Common pools of “hot” VMs
Some operations can be run in parallel	→	Actually all @op-function calls are lazy :)

Performance

Utilization ~95-99%

Median operation startup time ~10 sec

*allocated time/code running time ratio

The very last slide… almost

Business → Costs optimization

DataScientists → Friendly UX

Costs optimization + Friendly UX → Serverless DS

Leave your feedback!

We have a lot of interesting problems...

Telegram: @tyooma

Co-organizer